03 AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

WebAssembly programs are produced ahead of time through a process that begins with the source code of the program and moves through a compiler, resulting in a WebAssembly program. Software Diversification can be achieved at any of these stages. Diversifying at the source code stage, however, is not practical due to the need of creating a distinct diversifier for each language compatible with WebAssembly. In contrast, focusing on the compiler stage presents a viable option, especially considering that 70% of WebAssembly binaries are created using LLVM-based compilers, as noted by Hilbig et al. [**?**]. Furthermore, implementing diversification at the WebAssembly program stage stands as the most generic strategy, applicable to any WebAssembly program in the wild. Therefore, this thesis focuses on the exploration of diversification strategies at the compiler and WebAssembly program stages, employing two main approaches: compiler-based and binary-based.

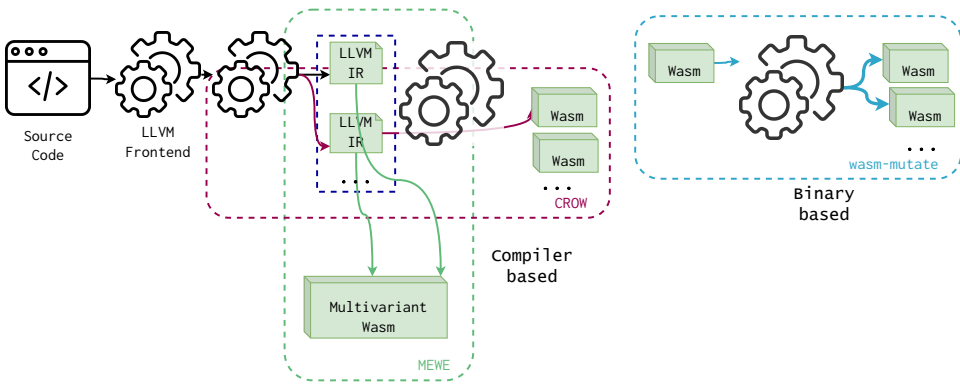

Figure 3.1: Approach landscape containing our three technical contributions: CROW squared in red, MEWE squared in green and WASM-MUTATE squared in blue. We annotate where our contributions, compiler-based and binary-based, stand in the landscape of generating WebAssembly programs.

Our compiler-based strategies are highlighted in red and green in Figure 3.1. This approach introduces a diversifier component in the LLVM pipeline, generating LLVM IR variants and producing artificial software diversity for Wasm. This strategy encompasses two tools: CROW [**?**], which creates Wasm program variants, and MEWE [**?**], which merges these variants to provide multivariant execution for Wasm. In contrast, the binary-based strategy, illustrated in blue in Figure 3.1, offers diversification for any WebAssembly program, removing the need for compiler tuning. WASM-MUTATE [**?**] generates a pool of WebAssembly program variants through rewriting rules upon an e-graph [**?**] data structure.

This dissertation contributes to the field of Software Diversification for WebAssembly, presenting three main technical contributions: CROW, MEWE, and WASM-MUTATE, dissected upon in the subsequent sections. Moreover, we compare our technical contributions between them. Furthermore, we outline the artifacts for our three technical contributions for the sake of open-research and reproducibility.

3.1 CROW: Code Randomization of WebAssembly

This section details CROW [**?**], represented as the red squared tooling in Figure 3.1. CROW is designed to produce semantically equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combining*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combining* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm binaries.

3.1.2 Variants' generation

The primary component of CROW's exploration process is its code replacement generation strategy. The diversifier implemented in CROW is based on the proposed superdiversifier methodology of Jacob et al. [**?**]. A superoptimizer focuses on *searching* for a new program that is faster or smaller than the original code while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [**?**] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations

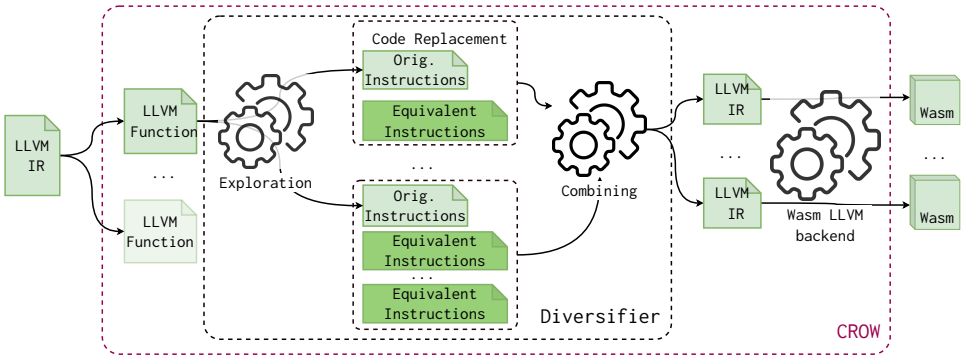

Figure 3.2: CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them. Figure taken from [**?**].

of optimized programs, sorted by code size in ascending order. If any of these programs is found to perform the same function as the source program, the search halts. On the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We build CROW upon an already existing superoptimizer for LLVM called Souper [**?**]. Yet, we modify it to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation is the optimal replacement result of the searching. CROW keeps more equivalent replacements during the searching by removing the halting criteria. Instead of the original halting conditions, CROW does not halt when it finds the first replacement. CROW continues the search until a timeout is reached or the replacements grow to a size larger than a predefined threshold.

Notice that the searching space increases exponentially with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions without correspondence in the Wasm backend. This decision reduces the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more program variants. For example, Souper prevents the generation of commutative operations during the searching. On the contrary, CROW still uses such transformation as a strategy to generate program variants.

The last stage involves the custom Wasm LLVM backend, which generates the Wasm programs. For it, we have the premise of removing all built-in optimizations in the LLVM backend that could reverse Wasm variants. We disable all optimizations in the Wasm backend that could reverse the CROW transformations.

3.1.3 Constant inferring

CROW, through using Souper adds a new transformation strategy that leads to more Wasm program variants, *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on variables that are used to control branches. After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

```
1  int babbage() {
2      int current = 0,
3          square;
4      while ((square=current*current) %
           ↪ 1000000 != 269696) {
5          current++;
6      }
7      printf ("The number is %d\n",
           ↪ current);
8      return 0 ;
9  }
```

Listing 3.1: Babbage problem. Taken from [**?**].

```
1  int babbage() {
2      int current = 25264;
3
4
5
6
7
8
9      printf ("The number is %d\n", current);
10     return 0 ;
11 }
```

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1. Taken from [**?**].

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the `while`-loop because the loop count is too large. The original Souper deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[1]

[1]Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR.

3.1.4 Combining replacements

When we retarget Souper, to create variants, we recombine all code replacements, including those for which a constant inferring was performed. This allows us to create variants that are also better than the original program in terms of size and performance. Most of the Artificial Software Diversification works generate variants that are as performant or iller than the original program. By using a superdiversifier, we could be able to generate variants that are better, in terms of performance, than the original program. This will give the option to developers to decide between performance and diversification without sacrificing the former.

On the other hand, when Souper finds a replacement, it is applied to all equal instructions in the original LLVM binary. In our implementation, we apply the transformations one by one. For example, if we find a replacement that is suitable for N difference places in the original program, we generate N different programs by applying the transformation in only one place at a time. Notice that this strategy provides a combinatorial explosion of program variants as soon as the number of replacements increases.

3.1.5 CROW instantiation

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

```
1  int f(int x) {
2      return 2 * x + x;
3  }
```

Listing 3.3: C function that calculates the quantity $2x + x$.

```
define i32 @f(i32) {
  %2 = mul nsw i32 %0,2
  %3 = add nsw i32 %0,%2

  ret i32 %3
}

define i32 @main() {
  %1 = tail call i32 @f(
       i32 10)
  ret i32 %1
}
```

```
Replacement candidates
      for code_1
%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2
%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

Listing 3.4: LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

Listing 3.5: Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping*.

```
func $f (param i32) (result i32)
   local.get 0
    i32.const 2
    i32.mul
    local.get 0
    i32.add

func $f (param i32) (result i32)
    local.get 0
    local.get 0
    i32.add
    local.get 0
    i32.add
```

```
func $f (param i32) (result i32)
    local.get 0
    i32.const 1
    i32.shl
    local.get 0
    i32.add

func $f (param i32) (result i32)
    local.get 0
    i32.const 3
    i32.mul
```

Listing 3.6: Wasm program variants generated from program Listing 3.3.

One might think that a reasonable heuristic could be implemented to avoid such overlapping cases. Instead, we have found it easier and faster to generate the variants with the combination of the replacement and check their uniqueness after the program variant is compiled. This prevents us from having an expensive checking for overlapping inside the CROW code. Still, this phenomenon calls for later optimizations in future works.

Contribution paper and artifact

CROW fully presented in Cabrera-Arteaga et al. "CROW: Code Randomization of WebAssembly" *Network and Distributed System Security Symposium, MADWeb* `https://doi.org/10.14722/madweb.2021.23004`.

CROW source code is available at `https://github.com/ASSERT-KTH/slumps`

3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [**?**]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE). MEWE generates application-level multivariant binaries without changing the operating system or Wasm runtime. It creates an MVE by intermixing functions for which CROW generates variants, as illustrated by the green square in Figure 3.1. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

In Figure 3.3, we focus on MEWE, highlighted in green in Figure 3.1. MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE merges the LLVM IR variants created by CROW and creates an LLVM multivariant binary. The merging of the variants intermixes the calling of function variants, allowing the execution path randomization.

The Mixer augments the LLVM multivariant binary with a random generator. The random generator is needed to perform the execution-path randomization. Also, *The Mixer* fixes the entrypoint in the multivariant binary. Finally, using the same custom Wasm LLVM backend as CROW, MEWE generates a standalone multivariant Wasm binary. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

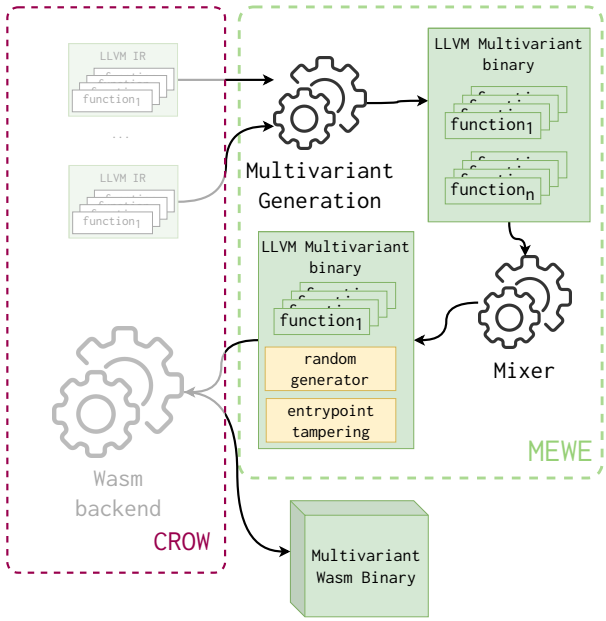

Figure 3.3: Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed. Figure partially taken from [**?**].

3.2.2 Multivariant generation

The key component of MEWE consists of combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The

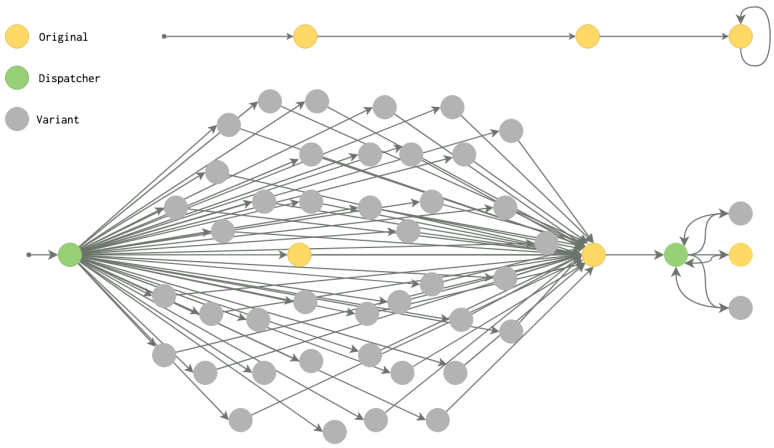

Figure 3.4: Example of two static call graphs. At the top, is the original call graph, and at the bottom, is the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow). Figure taken from [**?**].

directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.7: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4.

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher

corresponding to the right most green node of Figure 3.4. It first calls the random generator, which returns a value used to invoke a specific function variant. We utilize a switch-case structure in the dispatchers to prevent indirect calls, which are vulnerable to speculative execution-based attacks [**?**]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [**?**]. This also allows MEWE to inline function variants inside the dispatcher instead of defining them again.

MEWE has four specific objectives: link the LLVM multivariant binary, inject a random generator, tamper the application's entrypoint, and merge all these components into a multivariant Wasm binary. We use the Rustc compiler[2] to orchestrate the mixing. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. The Mixer component of MEWE creates a new entrypoint for the binary called *entrypoint tampering*. It wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

Contribution paper and artifact

MEWE is fully presented in Cabrera-Arteaga et al. "Multi-Variant Execution at the Edge" *Conference on Computer and Communications Security, MTD* `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

MEWE is also available as an open-source tool at `https://github.com/ASSERT-KTH/MEWE`

3.3 WASM-MUTATE: Fast and Effective Binary for WebAssembly

In this section, we introduce our third technical contribution, WASM-MUTATE [**?**], a tool that generates functionally equivalent variants of a WebAssembly program input. Leveraging rewriting rules and e-graphs [**?**] for diversification space traversals, WASM-MUTATE synthesizes program variants by altering parts of the original binary. In Figure 3.1, we highlight WASM-MUTATE as the blue squared tooling for a visual representation.

Figure 3.5 illustrates the workflow of WASM-MUTATE, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g. an intermediate representation. Subsequently,

[2] `https://doc.rust-lang.org/rustc/what-is-rustc.html`

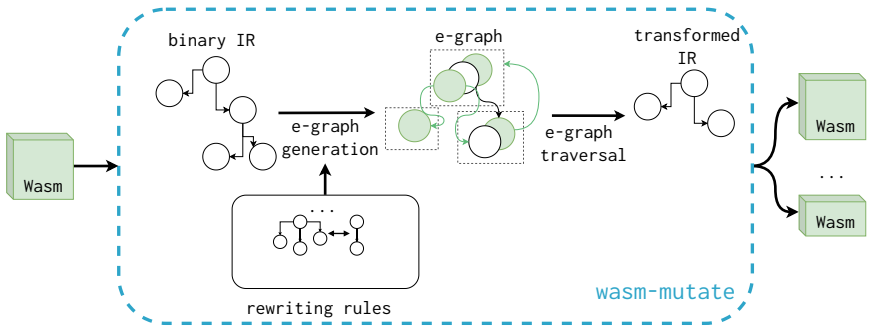

Figure 3.5: WASM-MUTATE high-level architecture. It generates semantically equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by diversification space traversals using e-graphs.

WASM-MUTATE utilizes predefined rewriting rules to construct an e-graph for the initial program, encapsulating all potential equivalent codes derived from the rewriting rules. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains functional equivalence to the original binary. This assurance of semantic preservation is rooted in the inherent properties of the individual rewrite rules employed.

3.3.2 WebAssembly Rewriting Rules

WASM-MUTATE incorporates a total of 135 rewriting rules, organized into categories referred to as meta-rules. The rewriting rules are conceived based on the seminal work of Sasnauskas et al. [**?**], extended to include a predicate to enforce the conditions for replacement. Each rule is articulated as a tuple, represented as `(LHS, RHS, Cond)`, where: `LHS` identifies the segment of code targeted for replacement, `RHS` outlines the functionally equivalent substitute, and `Cond` defines the circumstances permitting the replacements. For example, in the case of WebAssembly binaries, the `Cond` predicate ensures that the replacement does not violate the type constraints. The following text details the seven meta-rules utilized in WASM-MUTATE.

Add type: WASM-MUTATE implements two rewrite rules for the Type Section of the input WebAssembly program. These rewriting rules create random function signatures, varying both, the number of parameters and the count of results. It ensures the consistency of the index of already defined types, even after the introduction of a new type. The following listing illustrates how WASM-MUTATE adds a new type definition based on this meta-rule.

```
LHS (module
  (type (;0;) (func (param i32) (result i64)))

---

RHS (module
  (type (;0;) (func (param i32) (result i64)))
  (type (;0;) (func (param i64 ... ) (result i32 ... )))
```

Add function: WASM-MUTATE adds new random functions by mutating the code, type, and function sections. This process begins with the creation of a random type signature, followed by the formulation of a random function body that simply returns the default value corresponding to the result type. An illustration of this transformation is provided in the subsequent example.

```
LHS (module
   (type (;0;) (func (param i32 f32) (result i64)))

---

RHS (module
   (type (;0;) (func (param T) (result t)))
   (func (;0;) (type 0) (param T) (result t)
     t.const 0)
```

Debloat: WASM-MUTATE randomly eliminates dead parts of the input Wasm program, targeting specific elements such as *functions, types, custom sections, imports, tables, memories, globals, data segments, and elements* that are verifiably unused. For instance, the removal of a memory declaration needs the absence of any memory access operations within the binary code. WASM-MUTATE incorporates distinct mutators for each element type to facilitate this process. The following example showcases a function removal using this meta-rule.

```
LHS (module (import "" "" (func ))))

---

RHS (module )

Cond The removed function is not called, it is not exported, and it is not
     in the binary table.
```

Edit custom sections: WASM-MUTATE randomly changes either the content or the name of the custom section, a process illustrated in the subsequent example.

```
LHS (module
...
      (@custom "CS42" "zzz..."
```

```
RHS (module
...
       (@custom "..." "...")
```

If swapping: In WebAssembly, the if-construction is a compound of two paths: the consequence and the alternative. The determination of which execution path to follow depends on the branching condition evaluated just before the `if` instruction. Specifically, a value greater than `0` at the top of the execution stack triggers the execution of the consequence code, while any other outcome initiates the alternative code. The *if swapping* rewriting rule interchanges the consequence and alternative codes within the if-construction, effectively reversing the original paths defined by the condition.

To facilitate the swapping of an if-construction in WebAssembly, WASM-MUTATE introduces a negation of the value situated at the top of the stack immediately preceding the `if` instruction. The methodology behind this rewriting is demonstrated in the following example.

```
LHS (module
    (func ...) (
 condition C
       (if A else B end)
    )
)
```

```
RHS (module
    (func ...) (
 condition C
i32.eqz
       (if B else A end)
    )
)
```

In this context, the consequence and alternative codes are labeled with the letters `A` and `B`, respectively, while the if-construction's condition is represented as `C`. To negate this condition, the `i32.eqz` instruction is incorporated into the RHS segment of the rewriting rule, functioning to compare the stack's top value with zero and, if true, pushing the value `1` onto the stack. In addition, WASM-MUTATE introduces a `nop` instruction to substitute for the absent code block, ensuring a seamless rewriting process.

Loop Unrolling: WASM-MUTATE randomly unrolls loops. To unroll a loop WASM-MUTATE first creates a new Wasm block, which contains a copy of its

body (unrolling) followed by the original loop. The copy of the loop body is itself a Wasm block. To maintain the original control flow functionality, the instructions inside the loop and their copied body need to be adjusted. To adjust the loop, WASM-MUTATE modifies the loop instructions that are first-order breaks, i.e., jumps that lead back to the loop's beginning and end (see section Subsection 2.1.4).

Inside the loop's body, there can be two types of first-order breaks: the first type which leads back to the loop's beginning, and the second type jumps, which leads to the loop's end. The second type is irrelevant for the unrolling process since the loop's end is not modified. To adjust first-order breaks, in the case of the copied body, they need to break the Wasm block that contains the loop body copy, making the execution of the program continue with the loop's original construction appended after it. In the case of the original loop, the first-order breaks need to interrupt the block that contains the loop body, making the execution of the program to continue as originally as the loop finishes. In concrete, their jumping indexes need to be incremented by one, going outside the loop-unrolling outer Wasm block. In the following example, we illustrate the unrolling of a loop.

```
LHS (module
    (func ...) (
        (loop   A br_if 0   B end)
  )
)
```

```
RHS (module
    (func ...) (
        (block
            (block A'   br_if 0   B'   br 1   end)
            (loop   A'   br_if 0   B'   end)
        end)
    )
)
```

In the LHS part of the rewriting rule, the loop showcases the first-order break. The loop concludes just before the `end` instruction if the break is not triggered. When WASM-MUTATE unrolls this loop, it undergoes a bifurcation of its instructions into two distinct groups, `A` and `B`. The RHS part of the illustration creates the two fresh Wasm blocks used for unrolling. Here, the outer block is a container for both the original and the duplicated loop body, while the inner entities, labeled `A'` and `B'`, embody adjustments to the jump directives originally found in groups `A` and `B`. Moreover, the conclusion of the unrolled loop body copy is marked by the insertion of an unconditional branch `br 1`. This strategic placement guarantees that, in the absence of a continuation in the loop body, the operation exits the scope.

Peephole: This meta-rule focuses on the rewriting of instruction sequences

found within function bodies, representing the lowest level of rewriting. In WASM-MUTATE, we have devised 125 rewriting rules specifically for this category. WASM-MUTATE is structured to ensure the determinism of the instructions selected for replacement. Therefore, any rewriting rule inside the Peephole meta-rule avoids instructions that might induce undefined behavior, e.g., function calls. Consequently, the scope of this meta-rule is confined to modifications in stack and memory operations, preserving the original functionality of the control frame labels.

The peephole category rewriting rules are meticulously designed and manually verified. An instance of a rewriting rule in this category can be appreciated below:

```
LHS (x)
```

```
RHS (x i32.or x)

Cond x is i32 type
```

The previous rewriting rule example implies that the `LHS` 'x' is to be replaced by an idempotent bitwise `i32.or` operation with itself, as soon as x, which can be any subexpression, leaves a value of type i32 in the execution stack.

3.3.3 Extending peephole meta-rules with custom operators

As illustrated in Figure 3.5, the initial step in the process involves parsing an input WebAssembly program, generating an intermediate representation. This step facilitates the transition of the WebAssembly program to the next stages of WASM-MUTATE. This representation extends the textual Wat format. We augment it with custom operator instructions to enhance the transformation capabilities of WASM-MUTATE.

Custom operator instructions form part of the lowest level of transformation we provide in WASM-MUTATE, the Peephole meta-rule. These custom operator instructions are designed to bolster WASM-MUTATE by utilizing well-established code diversification techniques through rewriting rules. WASM-MUTATE includes four custom operator instructions in its intermediate representation of Wasm programs. In the following text, we describe each one of them. We also show concrete rewriting rules in the Peephole meta-rule that use them.

container : it acts as a holder for multiple instructions, enabling transformations without altering the program's semantics. Below, we demonstrate a rewriting rule that leverages it to insert `nop` instructions into the any WebAssembly program place, a well-known low-level diversification strategy [**?**]:

```
LHS x

---

RHS (container (x nop))
```

The instruction `x` (it can be a complete subexpression), can be substituted with `container (x nop)`. Thus, when converting back the intermediate representation to Wasm, a `nop` opcode is appended to the original instruction.

`useglobal:` this operator is used in a rewriting rule that substitutes its operand with the setting and retrieval actions involving a newly created global variable. In the following listing, we illustrate such a rewriting rule.

```
LHS x

---

RHS (useglobal x)
```

This rewriting rule is meant to stress the managed memory through random access to global variables.

unfold: this operator, working with a constant numeric operand, statically generates two numbers whose sum equals the constant, followed by the addition of operations for these numbers.

```
LHS (i32.const x)

---

RHS (unfold x)
```

rand: this operator injects random constant numbers in any place of the program. One of the rewriting rules using this operator is shown below.

```
LHS x

---

RHS (container ( x drop (rand) ))
```

In this rewriting rule, a subexpression is replaced by a container for which operands are the subexpression itself and the pushing of a random value into the execution stack, to be removed after with a `drop` instruction.

In practice, custom operators are only part of the rewriting rules of WASM-MUTATE. This means that, when converting Wasm to the intermediate representation no custom operator is generated. When converting back to the WebAssembly binary format from the intermediate representation, custom instructions are meticulously handled to retain the original functionality of the WebAssembly program. For example, the `container` custom operator is removed while its operands are encoded back to Wasm in their corresponding opcodes.

3.3.4 E-graphs

We developed WASM-MUTATE leveraging e-graphs, a specific graph data structure for representing rewriting rules [**?**]. Within an e-graph, there exist two distinct node types: e-nodes and e-classes. The former encapsulates either an operator or an operand present in the rewriting rule, while the latter groups e-nodes into equivalence classes, essentially serving as a composite virtual node that contains a collection of e-nodes. Consequently, each e-class contains at least one e-node. e-nodes have edges delineating the operator-operand equivalence relations between e-classes.

In the context of WASM-MUTATE, the e-graph is constructed from a WebAssembly program. This entails the transformation of each distinct expression, operator, and operand into e-nodes. A primer e-graph is built from the original program. This initial e-graph is subsequently augmented with e-nodes and e-classes derived from each one of the rewriting rules.

Let us illustrate the e-graph construction with a program that consists of a single instruction that returns an integer constant, denoted as `i64.const 0`. Besides, assume a single rewriting rule defined as `(x, x i64.or x, x returns type i64)`. In Figure 3.6 we visualize how the e-graph is built out of the program and rewriting rule.

Building the e-graph begins with the incorporation of the solitary program instruction, `i64.const 0`, as an e-node ①. Then, we derive additional e-nodes from the rewriting rule ②. This involves introducing a fresh e-node labeled `i64.or` and establishing edges leading to the `x` e-node. Since there is no extra condition in which the operator-operand relation is valid, `x` represents any e-class in the e-graph. Thus, the equivalence by merging the two e-nodes is affirmed, thus forming a unified e-class ③. Finally, we successfully construct an e-graph that encapsulates the relationships and equivalences dictated by the initial program and the rewriting rule ④, setting the stage for further analyses and transformations based on this structured representation.

3.3.5 Random e-graph traversal for variants generation

Willsey et al. demonstrated the potential for high flexibility in extracting code fragments from e-graphs, a process that can be recursively orchestrated through a cost function applied to e-nodes and their respective operands. This methodology ensures the semantic equivalence of the derived code [**?**]. For instance, e-graphs solve the problem of providing the best code out of several optimization rules [**?**]. To extract the "optimal" code from an e-graph, one might commence the extraction at a specific e-node, subsequently selecting the AST with the minimal size from the available options within the corresponding e-class's operands. In omitting the cost function from the extraction strategy leads us to a significant property: *any path navigated through the e-graph yields a semantically equivalent code variant.*

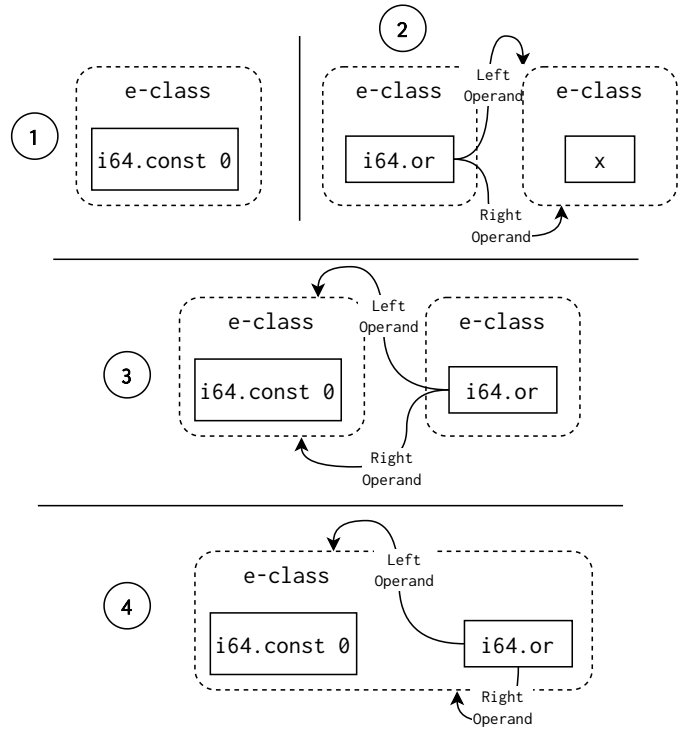

Figure 3.6: e-graph construction for idempotent bitwise-or rewriting rule and a single instruction Wasm program. Solid lines represent operand-operator relations, and dashed lines represent equivalent class inclusion.

The previously mentioned property is exploited in Figure 3.6, showcasing the feasibility of crafting an endless series of "or" operations. We exploit such property to generate diverse WebAssembly variants. We propose and implement an algorithm that facilitates the random traversal of an e-graph to yield semantically equivalent program variants, as detailed in Algorithm 1. This algorithm operates by taking an e-graph, an e-class node (starting with the root's e-class), and a parameter specifying the maximum extraction depth of the expression, to prevent infinite recursion. Within the algorithm, a random e-node is chosen from the e-class (as seen in lines 5 and 6), setting the stage for a recursive continuation with the offspring of the selected e-node (refer to line 8). Once the depth parameter reaches zero, the algorithm extracts the most concise expression available within the current e-class (line 3). Following this, the subexpressions are built (line 10) for each child node, culminating in the return of the complete expression (line 11).

Algorithm 1 e-graph traversal algorithm taken from [**?**].

```
1: procedure TRAVERSE(egraph, eclass, depth)
2:     if depth = 0 then
3:         return smallest_tree_from(egraph, eclass)
4:     else
5:         nodes ← egraph[eclass]
6:         node ← random_choice(nodes)
7:         expr ← (node, operands = [])
8:         for each child ∈ node.children do
9:             subexpr ← TRAVERSE(egraph, child, depth − 1)
10:            expr.operands ← expr.operands ∪ {subexpr}
11:        return expr
```

3.3.6 WASM-MUTATE instantiation

Let us illustrate how WASM-MUTATE generates variant programs by using the before enunciated algorithm. Here, we use Algorithm 1 with a maximum depth of 1. In Listing 3.8 a hypothetical original Wasm binary is illustrated. In this context, a potential user has set two pivotal rewriting rules: `(x, container (x nop),)` and `(x, x i32.add 0, x instanceof i32)`. The initial rule, which has been previously discussed in Subsection 3.3.3, grants the ability to append a `nop` instruction to any subexpression within the program. Conversely, the latter rule articulates the equivalence of augmenting any numeric value by zero.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        i64.const 1)
)
```

Listing 3.8: Wasm function.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.add (
            i64.const 0
            i64.const 1
            nop
        ))
    )
```

Listing 3.9: Random peephole mutation using egraph traversal for Listing 3.8 over e-graph Figure 3.7. The textual format is folded for better understanding.

Leveraging the code presented in Listing 3.8 alongside the defined rewriting

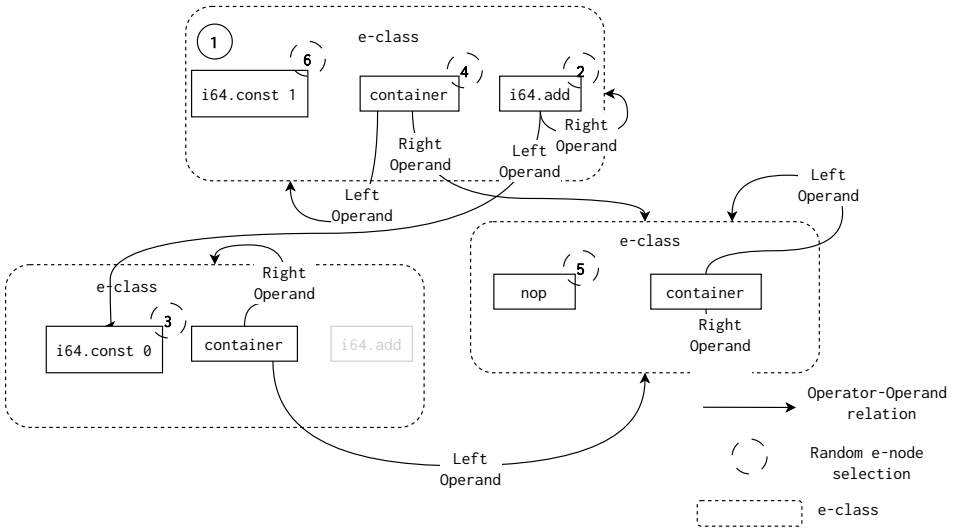

Figure 3.7: e-graph built for rewriting the first instruction of Listing 3.8.

rules, we build the e-graph, simplified in Figure 3.7. In the figure, we highlight various stages of Algorithm 1 in the context of the scenario previously described. The algorithm initiates at the e-class with the instruction `i64.const 1`, as seen in Listing 3.8. At ②, it randomly selects an equivalent node within the e-class, in this instance taking the `i64.add` node, resulting: `expr = i64.add l r`. As the traversal advances, it follows on the left operand of the previously chosen node, settling on the `i64.const 0` node within the same e-class ③. Then, the right operand of the `i64.add` node is choosen, selecting the `container` ④ operator yielding: `expr = i64.or (i64.const 0 container ( r nop ))`. The algorithm chooses the right operand of the `container` ⑤, which correlates to the initial instruction e-node highlighted in ⑥, culminating in the final expression: `expr = i64.or (i64.const 0 container( i64.const 1 nop)) i64.const 1`. As we proceed to the encoding phases, the `container` operator is ignored as a real Wasm instruction, finally resulting in the program in Listing 3.9.

Notice that, within the e-graph showcased in Figure 3.7, the container node maintains equivalence across all e-classes. Consequently, increasing the depth parameter in Algorithm 1 would potentially escalate the number of viable variants infinitely.

Contribution paper and artifact

WASM-MUTATE is fully presented in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" `https://arxiv.org/pdf/2309.07638.pdf`.

WASM-MUTATE is available at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate` as a contribution to the bytecodealliance organization [**?**].

3.4 Comparing CROW, MEWE, and WASM-MUTATE

In this section, we discuss the main differences between CROW, MEWE, and WASM-MUTATE. We discuss the main differences between CROW, MEWE, and WASM-MUTATE according to three main dimensions: 1) the technology and approach of each one, 2) the strength of the generated variants, and, 3) the security guarantees of the variants generated by each tool. We select these three dimensions because they lead the implementation of our tools.

3.4.2 Technology and approach

CROW is a compiler-based strategy, needing access to the source code or its LLVM IR representation to work. Its core is a Satisfiability Modulo Theories (SMT) solver, ensuring the functional equivalence of the generated variants. This approach lays the groundwork for a universal LLVM superdiversifier, potentially extending its applications and adaptability to other technologies. MEWE extends the capabilities of CROW, utilizing the same underlying technology to create program variants. It goes a step further by packaging the LLVM IR variants into a Wasm multivariant, providing MVE through execution path randomization.

On the other hand, WASM-MUTATE is a semi-automated, binary-based tool, centralizing its core around e-graph traversals. This approach facilitates the creation of a pool of WebAssembly program variants through the meticulous application of rewriting rules on an e-graph data structure. This method removes the need for compiler adjustments, offering compatibility with any existing WebAssembly binary. Moreover, it highlights how extending intermediate representations could establish a general framework for binary rewriting in WebAssembly.

3.4.3 Strength of the generated variants

CROW and MEWE use enumerative synthesis and verify semantic equivalence through SMT solvers. This approach not only has the potential to exceed

handcrafted optimizations but also ensures that the transformations are preserved. In other words, the transformations generated out of CROW and MEWE are virtually irreversible, even following compiler optimizations. This is particularly remarkable in the case of *constant inferring* transformations (see Subsection 3.1.3). While CROW and MEWE do not require any extra input but the program to diversify, the speed of variant generation is intrinsically linked to the SMT solvers' efficiency, known to be slow. Besides, their variants' generation capabilities are limited by the *overlapping* phenomenon discussed in Subsection 3.1.5.

On the other hand, WASM-MUTATE adopts a semi-automatic approach, requiring users to set the rewriting rules. Thus, the responsibility of ensuring functional equivalence is transferred to the rule creation process. This tool offers a significant advantage over CROW and MEWE as it permits transformations in any section of a Wasm program, not just the code section. Moreover, it leverages a virtually cost-free e-graph traversal process, avoiding, as a direct consequence, the *overlapping* issue seen in CROW and MEWE, as detailed in Subsection 3.1.5. In addition, since WASM-MUTATE operates at the binary level, it can modify functions incorporated by the WebAssembly producer itself. For example, this is the case of the *wasm32-wasi* architecture. While the original program might have a few lines of code, the underlying compiler might inject more functions to support the *wasm32-wasi* architecture. Thus, augmenting the diversification space available to WASM-MUTATE. Moreover, WASM-MUTATE outperforms CROW and MEWE capabilities in terms of the number of generated variants. Yet, the changes made by WASM-MUTATE might not be as preserved as the ones generated by CROW and MEWE. Thus, the variants generated by WASM-MUTATE might be more susceptible of being reversed, e.g. by further optimization passes.

Remarkably, CROW, MEWE, and WASM-MUTATE generate variants that potentially improve the original program's runtime performance, demystifying that software diversification inherently compromises performance.

3.4.4 Security guarantees

CROW and MEWE generate distinct and highly preserved code variants. This means that these variants, each with unique WebAssembly codes, maintain their distinctiveness even after JIT compilers translate them into machine codes (see Subsection 2.1.6). WASM-MUTATE, while offering slightly reduced preservation in its generated variants compared to CROW and MEWE, still maintains the same security guarantees excepting the multivariant cases. Its ability to produce a greater number of variants can offset this preservation shortfall. The preservation feature significantly reduces the impact of side-channel attacks that exploit specific machine code instructions, e.g., port contention [**?**].

Furthermore, CROW and MEWE enhance security against timing-based attacks by creating variants that exhibit a wide range of execution times.

This strategy is especially prominent in MEWE's approach, which develops multivariants functioning on randomizing execution paths, thereby thwarting attempts at timing-based inference attacks. Consequently, attackers might find it exceedingly difficult to identify a specific variant through time profiling of a MEWE multivariant (see Subsection 4.2.2 for the use case impact). Adding another layer benefit, the integration of diverse variants into multivariants can potentially disrupt dynamic analysis tools such as symbolic executors [**?**]. Concretely, different control flows through a random discriminator, exponentially increase the number of possible execution paths, making multivariant binaries virtually unexplorable.

An advantage of WASM-MUTATE, compared to CROW and MEWE, is its capacity to transform non-code sections without impacting the runtime behavior of the original variant, a strategy that effectively shields against static binary analysis, including malware detection based on signature sets [**?**]. For instance, it can modify the type section of a WebAssembly program, a section typically utilized only for function signature validation during compilation and validation processes by the host engine. This thwarts compiler identification techniques, such as fingerprinting. Besides, it can be used for masquerading as a different compilation source. Thus, reducing the fingerprinting surface available to attackers.

CROW, MEWE, and WASM-MUTATE can alter the original program structure, either by eliminating dead code or by introducing additional elements. From a static perspective, such alterations serve to reduce potential attack surfaces, thereby impeding signature-based identification. Yet, modifying the layout of a WebAssembly program inherently affects its managed memory during runtime, a segment not overseen by the WebAssembly program itself (see section **??** for a detailed discussion on unmanaged memory). This aspect is especially important for CROW and MEWE, given that they do not directly address the WebAssembly memory model. Significantly, CROW and MEWE considerably alter the managed memory by modifying the layout of the WebAssembly program. For example, the *constant inferring* transformations significantly alter the layout of program variants, affecting unmanaged memory elements such as the returning address of a function.

Furthermore, WASM-MUTATE not only affects managed memory through changes in the WebAssembly program layout. It also adds rewriting rules to transform unmanaged memory instructions, e.g. the rewriting rule involving the `useglobal` custom operator previously discussed in Subsection 3.3.3. Memory alterations, either to the unmanaged or managed memories, have substantial security implications. For instance, they can counteract attacks by eliminating potential jump points that facilitate malicious activities within the binary, a preventive measure highlighted by Narayan et al. [**?**].

3.5 Conclusions

In this chapter, we discuss the technical specifics underlying our primary technical contributions. We elucidate the mechanisms through which CROW generates program variants. Following this, we outline the conceptual framework for a universal LLVM superdiversifier, laying a foundation for broader applicability and versatility. Subsequently, we discuss MEWE, offering a detailed examination of its role in forging MVE for WebAssembly. We also explore the details of WASM-MUTATE, highlighting its pioneering utilization of an e-graph traversal algorithm to spawn Wasm program variants. Remarkably, we undertake a comparative analysis of the three tools, delineating their respective benefits and limitations, alongside the potential security assurances they provide upon the program variants derived from them.

In **??**, we present four use cases that support the exploitation of these tools. **??** serves to bridge theory with practice, showcasing the tangible impacts and benefits realized through the deployment of CROW, MEWE, and WASM-MUTATE.